華中科技大学

课 程 实 验 报 告

课程名称： 大数据系统综合实践

专业班级： 大数据 2101 班

学　　号： U202115652

姓　　名： 李嘉鹏

指导教师： 顾琳

报告日期： 2024 年 11 月 10 日

计算机科学与技术学院

目录

实验 1：大规模图数据中社区发现算法的设计与性能优化

1.1 实验概述

本实验旨在通过大规模图数据中社区发现算法的设计与性能优化，帮助学生深入理解图计算系统的工作原理和性能优化机制，并学会使用图计算框架进行大规模图数据分析和处理。通过此实验掌握图计算的基本原理，编写比较复杂的图算法程序并进行性能调优。

1.2 实验内容

社区发现是大数据社交网络分析中的一个重要领域，它旨在识别网络中具有紧密连接或共享相似特征的节点集合，这些集合被称为“社区”。

我们的生活中存在着各种各样的网络，如科研合作网络、演员合作网络、城市交通网络、电力网络，以及像 QQ、微博、微信这样的社交网络。社交网络的核心是参与其中的用户以及用户之间的关系。因此，我们可以采用图模型为其进行建模，其中的节点表示社交网络中一个个用户，而边则表示用户与用户之间的关系，如果想对这些关系强度（或亲密度）进行区分，我们还可以为每条边赋予一个权重，权值越大表示关系强度越大（或者越亲密）。如图 1 所示，这个图包含一定的结构：其中存在多个节点子集合，在这些子集合的内部边比较多，而子集合与子集合之间边比较少。将这些节点子集合圈出来，就会有三个内部联系非常紧密的社区。

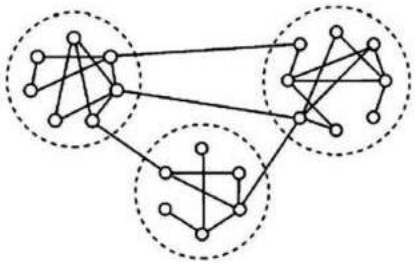

图 1：非重叠型（disjoint）社区

但是在某些场景下，节点子集合之间可能发生重叠，即同一个节点可能同时属于多个子集合。相应的图结构如图 2 所示，其中涂成彩色的节点表示同时参与了多个社区的关键节点。

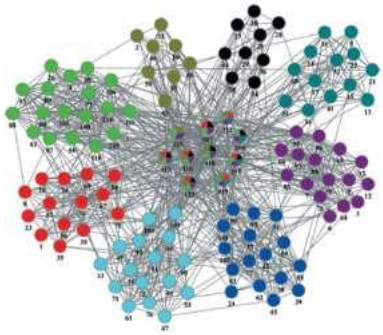

图 2：重叠型（overlapping）社区

那些内部连接比较紧密的节点子集合对应的子图叫做社区（community），各社区节点集合彼此没有交集的称为非重叠型（disjoint）社区，有交集的称为重叠型（overlapping）社区。网络图中包含多个社区的现象称为社区结构，社区结构是网络中的一个普遍特征。**现在，给定一个网络图，找出其社区结构的过程叫做社区发现（community detection）。**其主要有两种思路，分别为添加边或删除边。典型的社区发现算法包括 GN 算法（Girvan Newman）等。

本实验要求在给定数据集上实现社区发现算法，调试并获得最高的性能。在 Linux 环境下，采用 Spark GraphX 或 Pregel 或其它框架设计社区发现算法。所开发的算法需要充分利用多核资源，以完成给定格式的图数据中的社区发现并输出结果。开发环境如下：

- 操作系统：Linux ubuntu 14.04 或 16.04
- 编译器：gcc 或 g++ 4.8 以上
- Make：GNU make 4.0 以上
- 框架：Spark GraphX、Pregel 或你所熟悉擅长的其它框架

1.2.1 实验结果展示

本实验有功能和性能两方面的要求，首先需要正确地统计出所给数据集中的所有社区，其次需要不断优化算法（并行优化、存储优化、通信优化等），使得在最终的测试机器上执行时间最短。这一步需要在 CPU 上完成计算。

完成基础部分后，进阶部分要求在 GPU+CPU 的环境下，通过 GPU 加速来实现社区发现算法。

下面将分别给出优化后的算法在 CPU 和 GPU 两种环境下的运行结果。

（1）CPU 运行结果

①小数据集（如图 3）

名称	统计出的社区数量	程序运行时间
Cit-HepPh	82	17.6054 秒

```
社区 28: [9305318, 9207238, 9311377]
社区 30: [9810413, 9801389]
社区 31: [302100, 208137]
社区 33: [9401248, 9301251, 9408259]
社区 35: [205091, 104231]
社区 36: [104209, 9403375]
社区 39: [9505401, 9505374]
社区 41: [3062, 9912314]
社区 42: [9507392, 9305264]
社区 46: [301083, 212327]
社区 47: [206139, 5187]
社区 48: [9909272, 9809559]
社区 50: [4131, 9907529]
社区 51: [207088, 104222, 106003]
社区 53: [9403247, 9305249]
社区 68: [10242, 1265]
社区 69: [111036, 1277]

发现的社区总数: 82
程序执行时间: 17.6054 秒

Process finished with exit code 0
```

图 3：小数据集（CPU）运行结果

②大数据集（如图 4）

名称	统计出的社区数量	程序运行时间
Soc-LiveJournal1	5667	16681.2350 秒

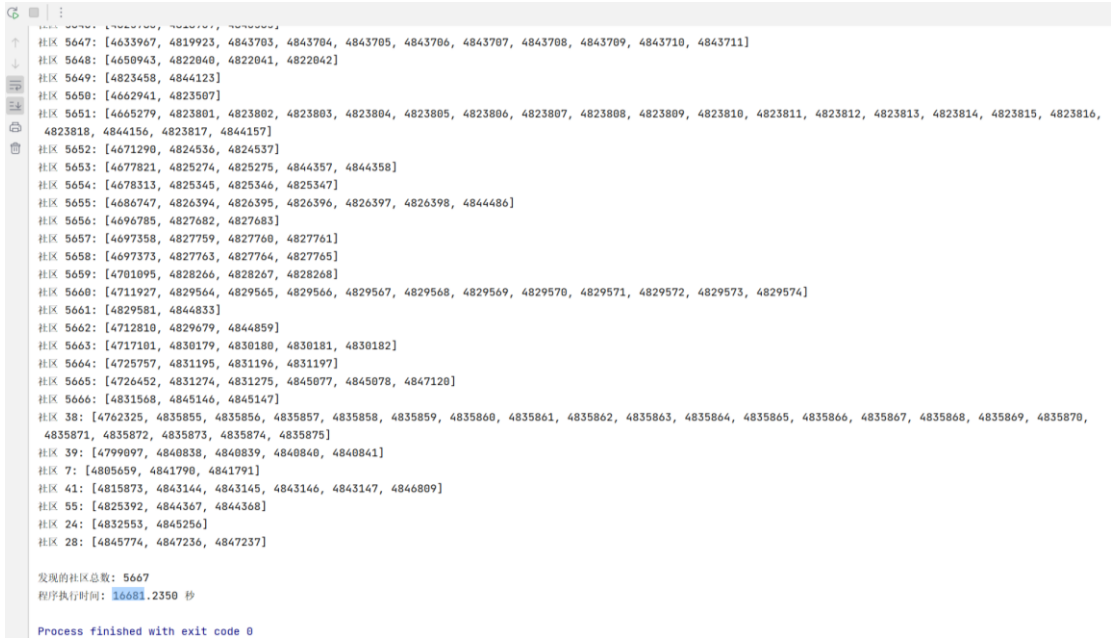

图 4：大数据集（CPU）运行结果

（2）CPU+GPU 加速运行结果

①小数据集（如图 5）

名称	统计出的社区数量	程序运行时间
Cit-HepPh	83	3.7321 秒

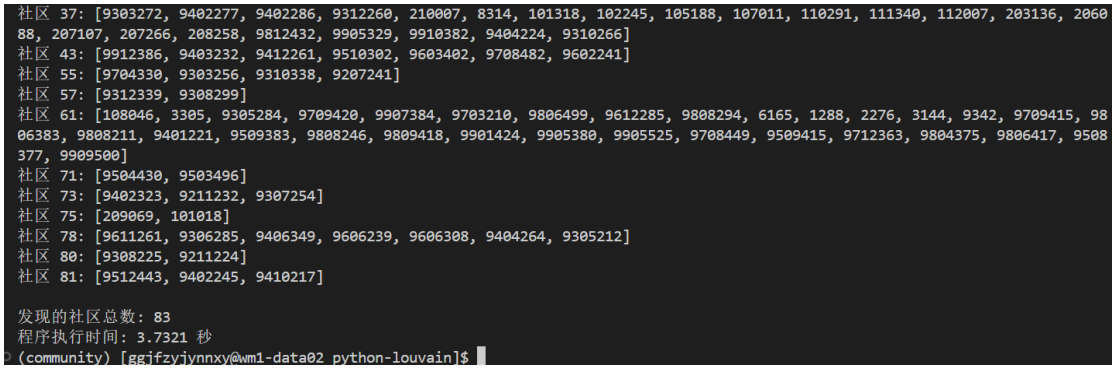

图 5：小数据集（CPU+GPU）运行结果

②大数据集（如图 6）

名称	统计出的社区数量	程序运行时间
Soc-LiveJournal1	6474	8.3592 秒

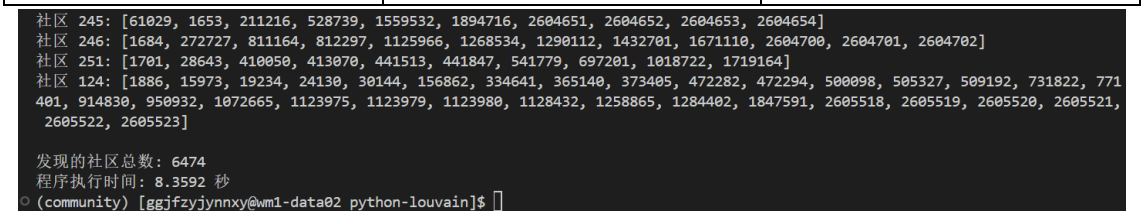

图 6：大数据集（CPU+GPU）运行结果

（3）CPU+GPU 相对于 CPU 运行的加速比

数据集/运行环境	CPU 运行时间	CPU+GPU 运行时间	加速比
小数据集	17.6054 秒	**3.7321 秒**	4.71
大数据集	16681.2350 秒	**8.3592 秒**	1995.55

1.2.2 算法实现与优化机制

1.主要算法实现

在本实验中，我先后尝试复现了 GN 算法和 Louvain 算法。

• GN 算法

GN 算法是基于删除边的算法，本质上是一种使用边介数（网络中任意两个节点通过此边的最短路径数量）作为相似度的度量方法。算法流程是：（1）计算

每条边的边介数；（2）删除边介数最大的边；（3）重新计算网络中剩下边的边介数；（4）重复（3）和（4）直到网络中的任一节点成为一个社区。

GN 算法作为社区发现领域的经典算法，**存在一些缺陷**，例如：（1）计算复杂度高，在每次删除边后都会重复计算剩余节点的边介数；（2）算法的终止条件缺乏一个可衡量的目标，无法控制最后可以分裂的社区数量（即社区的划分粒度）。如图 7 所示，GN 算法划分社区的过程本质上就是层次分裂的过程，相当于一棵自顶向下的层次树。

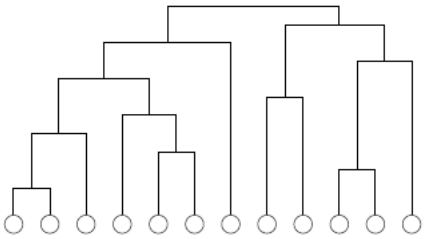

图 7：GN 算法的层次分裂过程

为了解决上述 GN 算法的缺陷，需要引入模块度 Q 来判定社区划分程度的好坏。模块度越高，说明检测到的社区越符合“内紧外松”的特征，分组质量越好。模块度的计算公式为：

$$Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j)$$

$$\delta(u,v) = \begin{cases} 1 & when\ u==v \\ 0 & else \end{cases} \qquad m = \frac{1}{2}\sum_{ij} A_{ij}$$

其中，A_{ij} 为节点 i 和 j 之间连接的权重，k_i 为节点 i 所有边的权重和，c_i 为节点 i 所属的社区。

对上面的 Q 作进一步变形可得：

$$Q = \frac{1}{2m}\sum_{i,j}[A_{ij} - \frac{k_i k_j}{2m}]\delta(c_i, c_j)$$

$$= \frac{1}{2m}[\sum_{i,j} A_{ij} - \frac{\sum_i k_i \sum_j k_j}{2m}]\delta(c_i, c_j)$$

$$= \frac{1}{2m}\sum_{c}[\Sigma in - \frac{(\Sigma tot)^2}{2m}]$$

其中，Σin 表示社区 c 内边的权重之和，Σtot 表示与社区 c 内节点相连的边的权重之和。

改进后的算法就是我最终采用的 Louvain 算法，详细介绍如下。

• Louvain 算法

Louvain 算法是一种无监督算法。输入一个包含 N 个节点的权重图，Louvain 算法分为以下两阶段，计算时会重复执行这两个阶段，直到模块度不再变化。

（1）模块度优化

输入权重图（初始时将每个节点视作一个社区），在所有节点上，随机选出节点 *i*，考虑节点 *i* 的邻居节点 *j*，通过将节点 *i* 从原本的社区移至节点 *j* 所在的社区，判断是否会产生模块度的增益，并选择增益最大的社区进行移动，如果没有产生正增益则不移动。将节点 *i* 移动至社区 *c* 带来的模块化增益 ΔQ 为：

$$\Delta Q = \left[\frac{\sum_{in}+k_{i,in}}{2m} - \left(\frac{\sum_{tot}+k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

整个阶段（1）在没有更多增益时结束，此时模块度达到最大，任何节点合并均不可能继续提高模块度。

（2）社区聚合

将阶段（1）结束后的社区合并为一个新的节点，社区内点的权重转化为新结点环的权重，社区间的权重转化为新结点边的权重。为了最大化全局模块度，将节点进行合并后，构建一个新的子图并返回阶段（1）开始下一轮计算。随着每一次节点的合并，后续每轮迭代中的节点数量会持续减少，因此 Louvain 算法的效率很高。

图 8 展示了一个简单的例子，在第一轮计算中，16 个节点被划分为 4 个社区，并被合并为 4 个新的节点（新的子图）。随后在第二轮计算中，算法收敛。

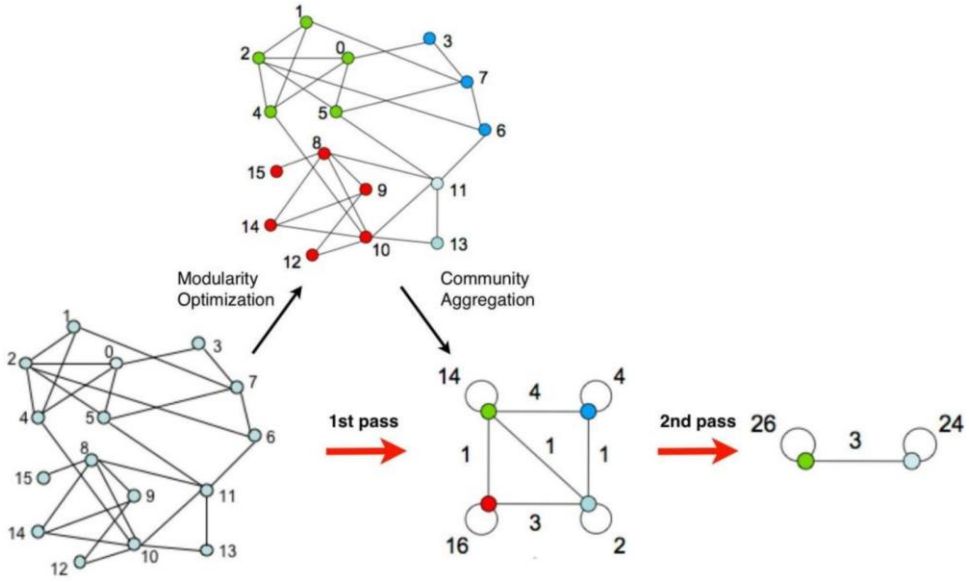

图 8：Louvain 算法的模块度优化和社区聚合过程

2.优化机制

在 Louvain 算法的基础上，我针对本任务的具体特点引入了三种优化方法。

（1）【并行优化】使用基于互斥锁的 threading 多线程，并行读取输入数据

①优化思路：由于数据集规模非常大，如果采用单线程串行读入的方式，则消耗时间过长。因此，可以考虑引入多个线程，它们各自负责读取和处理一部分数据，并统一合并到最终集合中。在本实验中，线程数量 parallel_threads 设置为 16。

②代码实现：

```
import threading

edges = []
n = 0
e = 0

# 使用互斥锁确保对共享数据的安全访问
lock = threading.Lock()

def process_line(line):
    global n, e
    u, v = map(int, line.split())
    with lock:
        edges.append((u, v))
        n = max(n, u, v)
        e += 1

def parallel_process(file_name, parallel_threads):
    threads = []
    with open(file_name, 'r') as file:
        for line in file:
            thread = threading.Thread(target=process_line, args=(line.strip(),))
            threads.append(thread)
            thread.start()
            if len(threads) == parallel_threads:
                for t in threads:
                    t.join()
                threads = []

    for t in threads:
        t.join()

parallel_threads = 16
parallel_process(file_name, parallel_threads)
```

（2）【并行优化】异步计算多个社区的聚合，并行产生新的子图

①优化思路：在 Louvain 算法的第二阶段，可以异步处理多个社区的聚合，也就是在处理一个社区并合并这部分节点的同时，其他社区节点的合并更新也能并行进行，从而提高整体效率（例如在图 8 中，可以同时独立合并 4 个社区的节点）。具体可以使用 asyncio 库来实现，下面的代码中，async_induced_graph 函数异步地产生新的子图，add_edge 函数异步地添加边。

②代码实现：

```
import asyncio

async def async_induced_graph(partition, graph, weight='weight'):
    """异步产生新的子图"""
    ret = nx.Graph()
    ret.add_nodes_from(partition.values())

    # 收集所有异步任务
    tasks = []
    for node1, node2, datas in graph.edges(data=True):
        edge_weight = datas.get(weight, 1)
        com1 = partition[node1]
        com2 = partition[node2]
        w_prec = ret.get_edge_data(com1, com2, {weight: 0}).get(weight, 0)
        tasks.append(asyncio.create_task(
            add_edge(ret, com1, com2, weight, w_prec + edge_weight)
        ))

    # 等待所有任务完成
    await asyncio.gather(*tasks)
    return ret

async def add_edge(graph, node1, node2, weight, edge_weight):
    """异步添加边"""
    graph.add_edge(node1, node2, **{weight: edge_weight})
```

（3）【存储优化】使用压缩稀疏行（CSR）数据结构储存图数据

①优化思路：CSR 被设计为仅储存非零元素，因此它特别适用于稀疏图的邻接矩阵表示。在 Louvain 算法的模块度优化和聚合阶段，利用预分配的 CSR 结构搜索社区的当前全部节点、遍历某节点的邻接节点和存储合并后新的节点，可以

大大提高存储和数据查询效率。

②代码实现：

```
import numpy as np
from scipy.sparse import csr_matrix

def create_csr(graph):
    """将图转换为 CSR 格式"""
    nrows = len(graph)
    col_indices = []
    row_ptr = [0]

    for node in graph:
        neighbors = graph[node]
        degree = len(neighbors)
        col_indices.extend(neighbors)
        row_ptr.append(row_ptr[-1] + degree)

    indices = np.array(col_indices, dtype=np.int32)
    # 创建图的权重数组
    data = np.ones(len(indices), dtype=np.int32)
    # 创建 CSR 矩阵
    return csr_matrix((data, indices, row_ptr), shape=(nrows, nrows))
```

1.3 实验总结

这是我在华中科技大学本科期间的最后一门实验课。在本次实验中，我主要基于Louvain算法实现了大规模数据集上的社区发现，同时针对任务的具体特性，从并行优化和存储优化角度尝试提高算法效率、降低程序运行时间。在 CPU 和 GPU 上的测试表明，优化后的算法可以在较短时间内得出合理的社区分类结果，证明实验中算法的实现和优化是有效的。

社区发现问题是网络分析中的核心问题，它需要在复杂网络中识别出高度相互连接的节点群体（被称为“社区”）。社区发现的目标是将网络中的节点划分成若干个社区，使社区内部节点之间比社区外部节点之间的连接更加密集。通过识别社区，可以更好地理解网络的结构和功能。然而，社区发现问题存在众多挑战，例如社区的大小和划分粒度没有统一标准，需要灵活适应不同的图结构，并且在某些超大规模网络中，节点可能同时属于多个社区，这增加了结果的复杂性。

Louvain 算法是基于模块度优化的无监督启发式算法，由于其计算速度快、收敛效果好、无需预先标注数据，因此是应用范围最广泛的社区发现算法之一。其基本思想是对每个节点尝试遍历其邻居所在的社区，并选择具有最大模块度增

量的社区进行移动。这一阶段结束后，每个社区将被合并，并且作为一个新的节点加入新的子图，开始下一轮迭代计算。最终，当全局模块度达到最大值时算法收敛，进而得到社区分类的结果。

在此基础上，我还竭尽所能通过各种优化手段加快程序的运行效率。具体来说，我设计了三种优化手段：

（1）在读取数据时引入互斥锁，从而允许了多线程并行读取并处理数据，最终合并得到完整的数据；

（2）在社区聚合阶段，我注意到多个社区可以同时完成节点合并操作，通过将串行逻辑修改为并行逻辑，可以极大提高社区聚合与更新的速度；

（3）最后，我从存储结构入手，选取了适合存放稀疏图数据的压缩稀疏行进行数据管理，提供了快速的行切片访问，这对于并行处理和社区发现中邻接节点的快速查找尤为重要。

综上所述，我从本次大数据系统综合实践课程中深入学习了如何构建一个能处理大规模数据的社区发现系统，显著提高了自己的代码与调试能力，丰富了项目经验。我认为在面对大规模数据时，精益求精、不放过任何一个细节才是真正负责的态度，极致的性能永远是我追求的目标，这也将成为我未来求学生涯中的长期指引。

在华科的这四年，时间煮雨，岁月逢花。最后，感谢各位老师过去的辛勤付出，也祝大数据专业未来能办的越来越好！